Student Id: 1001649826
Name: Pratik Barhate

CSE 5311
Project Report

I have implemented and compared the following sorting algorithms:

- Merge sort
- Heap sort
- Quick sort (Using median of 3)
- Insertion sort
- Bubble sort

The project report contains the implementation of all the algorithms stated above. It will cover main data structures, main components, inputs given to the algorithm and experimental results of the algorithms.
Each algorithm will be explained individually with results as well in order to cover explanation in detail. Results will contain the sorted array and also graph of 'input size vs running time' for 9 input arrays of varying lengths. All the algorithms are implemented using python language.

1. Merge sort

The algorithm for merge sort is as follows:

```
func mergeSort( var a as array )
if ( n == 1 ) return a
var l1 as array = a[0] ... a[n/2]
var l2 as array = a[n/2+1] ... a[n]
l1 = mergesort( l1 )
l2 = mergesort( l2 )
return merge( l1, l2 )
end func

func merge( var a as array, var b as array )
var c as array
while ( a and b have elements )
        if ( a[0] > b[0] )
                add b[0] to the end of c
                remove b[0] from b
        else
                add a[0] to the end of c
                remove a[0] from a

while ( a has elements )
        add a[0] to the end of c
        remove a[0] from a
while ( b has elements )
```

```
        add b[0] to the end of c
        remove b[0] from b
    return c
end func
```

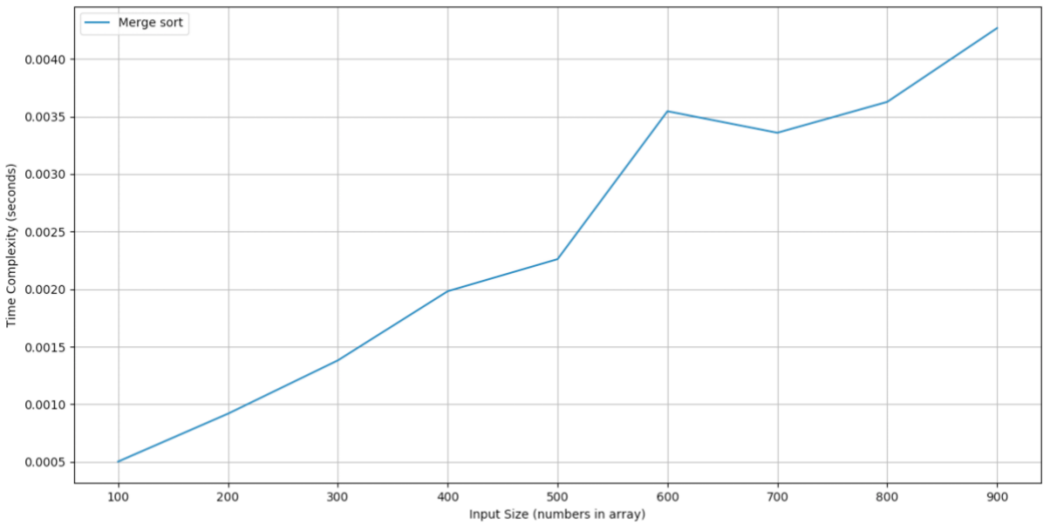
Merge sort is a divide-and-conquer algorithm based on the idea of breaking down a list into several sub-lists until each sub list consists of a single element and merging those sub lists in a manner that results into a sorted list.

1. Find the middle point to divide the array into two halves
2. Call mergeSort for first half
3. Call mergeSort for second half
4. Merge the two halves sorted in step 2 and 3

The data structures used for this algorithm implementation is Array and lists.
The merge(a,b) is key process that assumes that l1 and l2 are sorted and merges the two sorted sub-arrays into one.
The inputs provided for this algorithm code implementation is an Array with any size of choice as per user. The output of the code will be the same Array sorted by ASC.
The inputs provided for plotting the 'input size vs running time' graph of this algorithm is 9 random lists of size 100,200,300,400,500,600,700,800 and 900.

Merge sort requires space to create a new list of the same size as the input list.
The overall time complexity of the Merge Sort algorithm is O(nlog(n)).
Merge sort is not in-place algorithm and performs well with LinkedLists rather than Arrays.

The graph for merge sort algorithm is as follows:



From the graph we can deduce that as the input size increases the running time also increases which is normal and should happen.
For input size 100 the running time is 0.0005 seconds and for input size 900 the running time is 0.004 seconds.

2. **Heap sort**

The algorithm for heap sort is as follows:

```
Heapsort(A as array)
        BuildHeap(A)
        for i = n to 1
                swap(A[1], A[i])
                n=n- 1
        Heapify(A, 1)

BuildHeap(A as array)
        n = elements_in(A)
        for i = floor(n/2) to 1
                Heapify(A,i,n)

Heapify(A as array, i as int, n as int)
        left = 2i
```

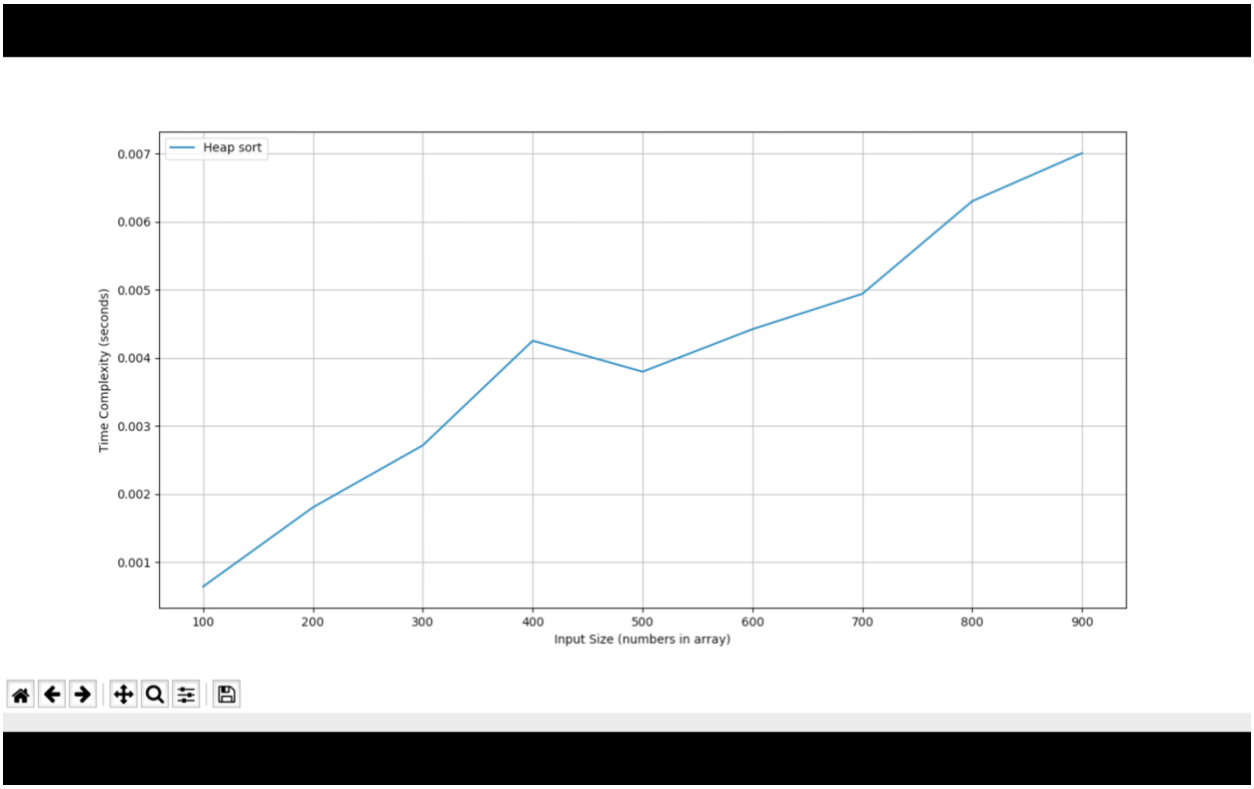
```
right = 2i+1
if (left <= n) and (A[left] > A[i]) max = left
else
max = i
if (right<=n) and (A[right] > A[max])
max = right
if (max != i)
swap(A[i], A[max])
Heapify(A, max)
```

In my code implementation I have not created a separate function of BuildHeap I have implemented the same logic under heapsort only. Also, here in heapify the left child is 2i and right child in 2i + 1 for a parent node i. But In code implementation since the list index starts from 0 I have taken left child as 2i + 1 and right child as 2i + 2 which gives the desired results.

Heap sort is a comparison-based sorting technique based on Binary Heap data structure.
1. Build a heap with the sorting array, using recursive insertion.
2. Iterate to extract n times the maximum or minimum element in heap and heapify the heap. 3. The extracted elements form a sorted subsequence.

The data structures used for this algorithm implementation is Array and lists.
Heapsort, even if all of the data is already sorted, the algorithm swaps all of the elements to order the array.
The inputs provided for this algorithm code implementation is an Array with any size of choice as per user. The output of the code will be the same Array sorted by ASC.
The inputs provided for plotting the 'input size vs running time' graph of this algorithm is 9 random lists of size 100,200,300,400,500,600,700,800 and 900.
The algorithm is efficient, and the performance is optimal. Memory usage is minimal because apart from what is necessary to hold the initial list of items to be sorted, it needs no additional memory space to work. The problem with heap sort is if the data set is too huge and doesn't fit into memory then there is an issue whereas merge sort works better in that case.
The overall time complexity of the Heap Sort algorithm is O(nlog(n)).

The graph for heap sort algorithm is as follows:



From the graph we can deduce that as the input size increases the running time also increases which is normal and should happen.
For input size 100 the running time is less than 0.001 seconds and for input size 900 the running time is 0.007 seconds.

3. Quick sort (Using median of 3)

The algorithm of quick sort (using median of 3) is as follows:

```
quick_sort(arr):
left = []
right = []
pivot_list = []
if len(arr) <= 1:
        return arr
else:
        pivot = arr[0]
for i in arr:
        if i < pivot:
                left.append(i)
        elif i > pivot:
```

```
                right.append(i)
        else:
                pivot_list.append(i)
left = quick_sort(left)
right = quick_sort(right)
return left + pivot_list + right
```

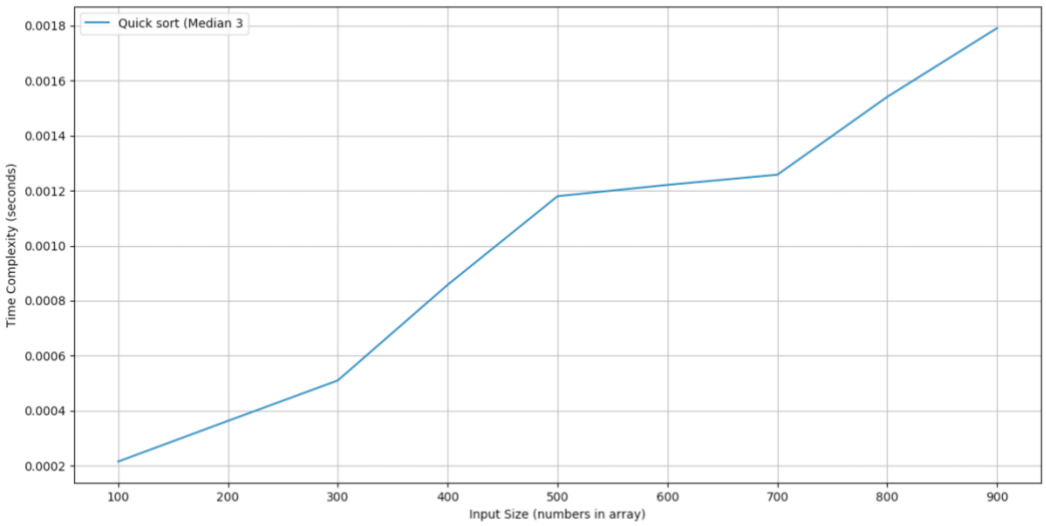
Quicksort sorts by employing a divide and conquer strategy to divide a list into two sub-lists.
The steps are:

1. Pick an element, call a pivot from the list.
2. Reorder the list so that all elements which are less than the pivot come before the pivot and so that all elements greater than the pivot come after it (equal values can go either way). After this partitioning, the pivot is in its final position. This is called the partition operation.
3. Recursively sort the sub-list of lesser elements and the sub-list of greater elements.

Quicksort with median-of-three partitioning functions nearly the same as normal quicksort with the only difference being how the pivot item is selected. In normal quicksort the first element is automatically the pivot item. This causes normal quicksort to function very inefficiently when presented with an already sorted list. The division will always end up producing one sub-array with no elements and one with all the elements (minus of course the pivot item). In quicksort with median-of-three partitioning the pivot item is selected as the median between the first element, the last element, and the middle element (decided using integer division of n/2). In the cases of already sorted lists this should take the middle element as the pivot thereby reducing the inefficiency found in normal quicksort.

In my code implementation the elements which are less than the pivot is kept in the left array and the elements which are greater than the pivot is kept in the right array.
The inputs provided for plotting the 'input size vs running time' graph of this algorithm is 9 random lists of size 100,200,300,400,500,600,700,800 and 900.
The inputs provided for this algorithm code implementation is an Array with any size of choice as per user. The output of the code will be the same Array sorted by ASC.
The overall time complexity of the quick Sort algorithm is O(nlog(n)).

The graph for quick sort (using median 3) algorithm is as follows:



From the graph we can deduce that as the input size increases the running time also increases which is normal and should happen.
For input size 100 the running time is about 0.0002 seconds and for input size 900 the running time is 0.0018 seconds.
Quicksort does not swap elements that are already in order, which is unnecessary and is done in heap sort. Hence, quick sort outperforms heap sort.
Also, the locality of reference in quicksort is better than merge sort. As a result, it outperforms merge sort as well.

4. **Insertion sort**

The algorithm of insertion sort is as follows:

Function to do insertion sort

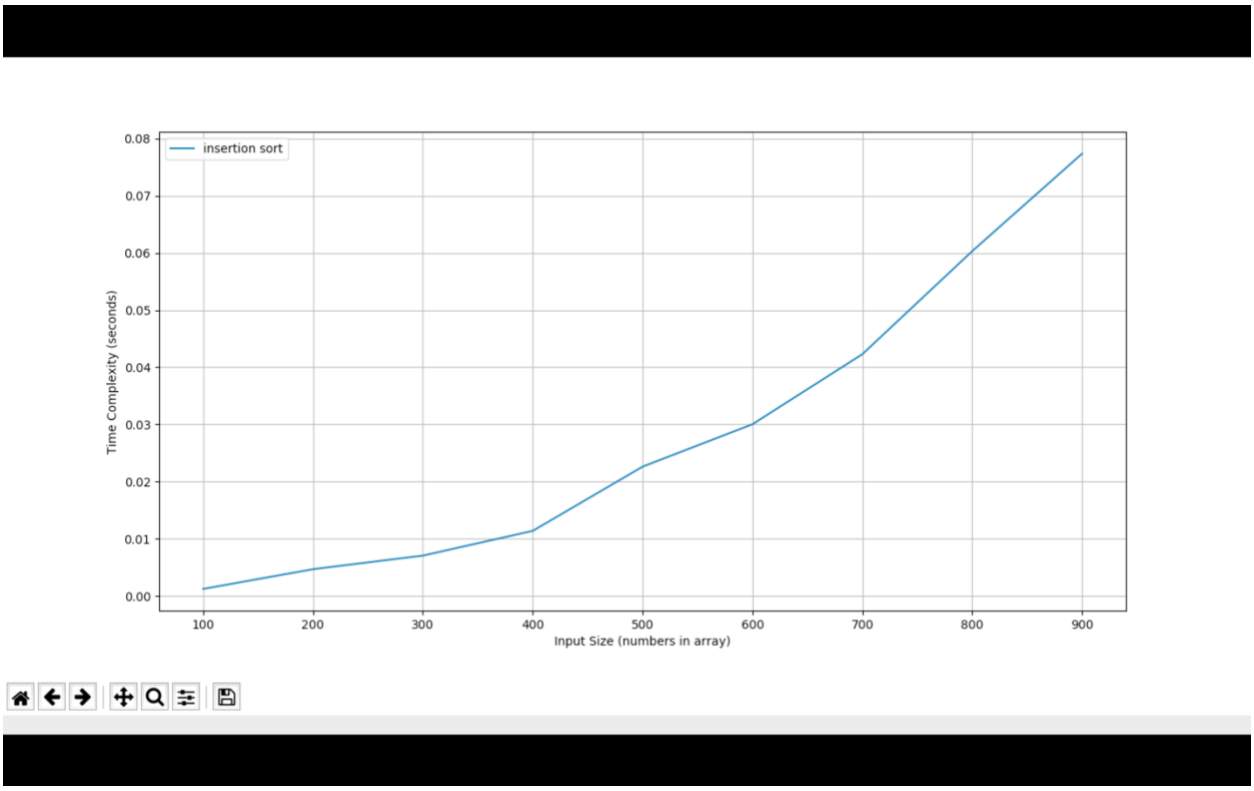
Function insertionSort(arr):
# Traverse through 1 to n for i from (1, n):
key = arr[i]
# Move elements of arr[0..i-1], that are
# greater than key, to one position ahead
# of their current position
j = i-1
while j >= 0 and key < arr[j] :

```
        arr[j + 1] = arr[j]
        j –
arr[j + 1] = key
```

Insertion sort is based on the idea that one element from the input elements is consumed in each iteration to find its correct position i.e., the position to which it belongs in a sorted array
Insertion sort moderate speed sorting algorithm for small n but not at all, for large n.
The main advantage of the insertion sort is its simplicity. It also exhibits a good performance when dealing with a small list. The insertion sort is an in-place sorting algorithm, so the space requirement is minimal. The disadvantage of the insertion sort is that it does not perform as well as the other sorting algorithms when the size of the data gets larger.

The inputs provided for plotting the 'input size vs running time' graph of this algorithm is 9 random lists of size 100,200,300,400,500,600,700,800 and 900.
The inputs provided for this algorithm code implementation is an Array with any size of choice as per user. The output of the code will be the same Array sorted by ASC.
The overall time complexity of the insertion Sort algorithm is $O(n^2)$.

The graph for insertion sort algorithm is as follows:



From the graph we can deduce that as the input size increases the running time also increases which is normal and should happen.
For input size 100 the running time is little above 0.00 seconds and for input size 900 the running time is almost 0.08 seconds.

As we can further conclude that with less input size the algorithm is processing at moderate speed but with the increase in data size there is drastic slowness in the running time.

5. **Bubble sort**

The algorithm of bubble sort is as follows:

```
Bubble_sort(array a )
while ( array a not sorted ):
        for( i = 0 ; i < a.length-1 ; i ++ )
                if ( a[i] > a[i+1] ) // Pair out of order
                        Swap a[i] and a[i+1]
```

This is the traditional approach of implementing a bubble sort. Bubble Sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in wrong order.
We begin by comparing the first two elements of the list. If the first element is larger than the second element, we swap them. If they are already in order, we leave them as it is. We then move to the next pair of elements, compare their values and swap as necessary. This process continues to the last pair of items in the list. Upon reaching the end of the list, it repeats this process for every item, and this is highly inefficient. To optimize the algorithm, we need to stop it when the sorting is finished. If the items were in order, then we would not have to swap items. So, whenever we swap values, we set a flag to True to repeat sorting process. If no swaps occurred, the flag would remain False and the algorithm would stop.

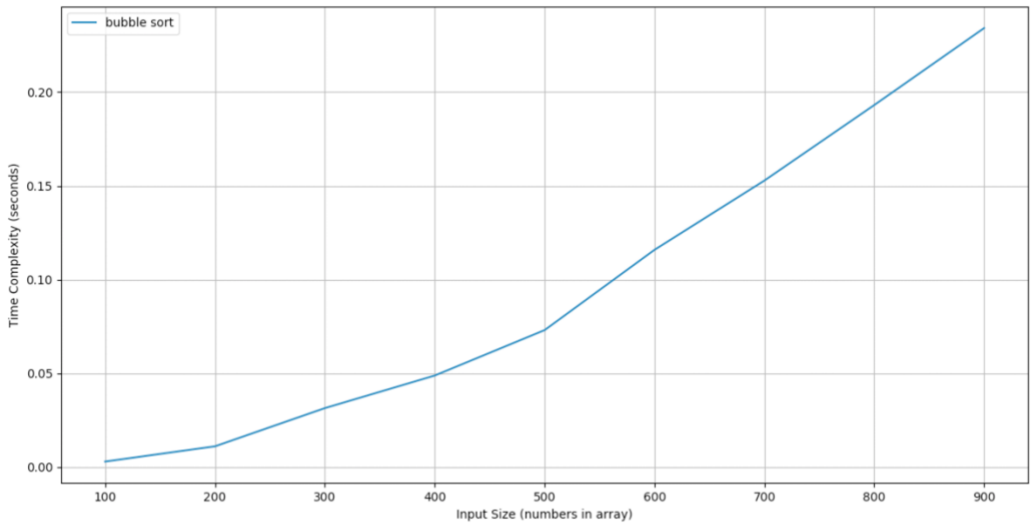
The optimized bubble sort algorithm is as follows:

```
bubble_sort(arr):
        length = len(arr) – 1
        sorted_arr = False
        while not sorted_arr:
                sorted_arr = True
                for i in range(length):
                        if arr[i] > arr[i + 1]:
                                sorted_arr = False
                                arr[i], arr[i + 1] = arr[i + 1], arr[i]
        return arr
```

In my code implementation the flag is the sorted_arr variable. The inputs provided for plotting the 'input size vs running time' graph of this algorithm is 9 random lists of size 100,200,300,400,500,600,700,800 and 900.
The inputs provided for this algorithm code implementation is an Array with any size of choice as per user. The output of the code will be the same Array sorted by ASC.
The overall time complexity of the bubble Sort algorithm is $O(n^2)$.
The advantages of bubble sort algorithm are that it is easy to understand, easy to implement, In-place sorting as a result no external memory is needed and performs very well when the array is mostly sorted.
The disadvantages of bubble sorting are very expensive as the running times are very high.

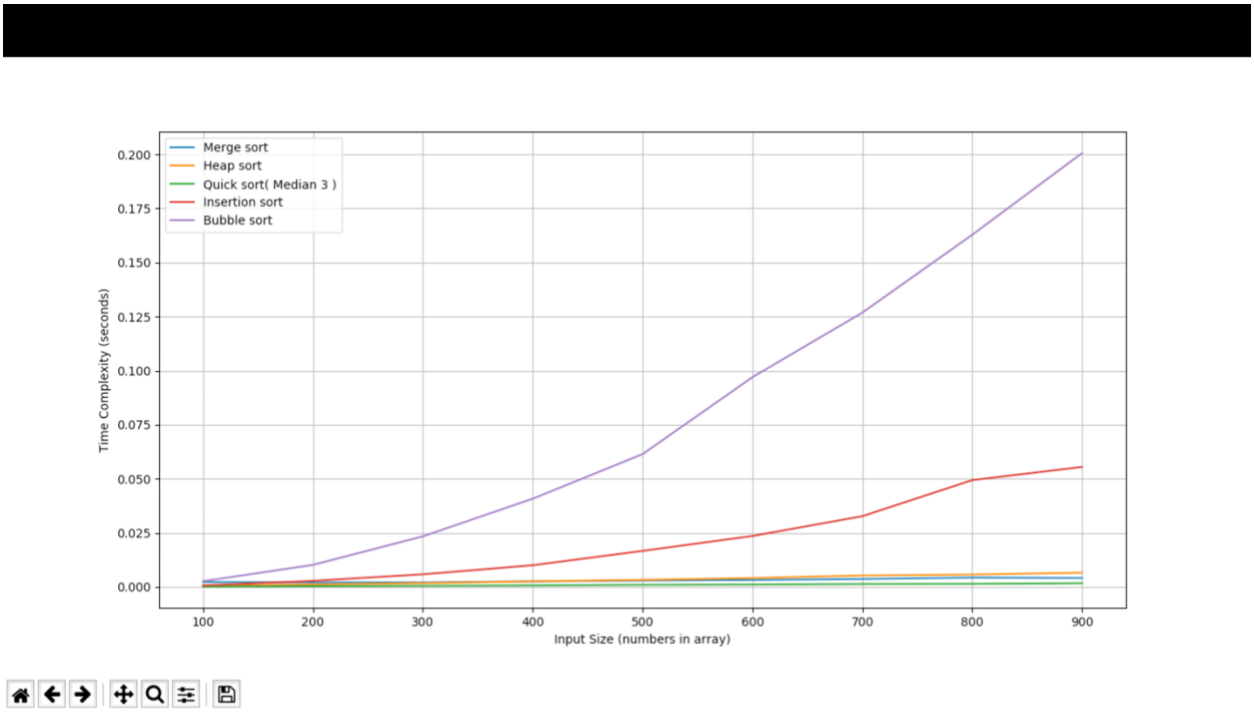
It does more element assignments than insertion sorting. Hence, Insertion sort outperforms bubble sort.

The graph for bubble sort algorithm is as follows:



From the graph we can deduce that as the input size increases the running time also increases which is normal and should happen.
For input size 100 the running time is little above 0.00 seconds and for input size 900 the running time is almost 0.30 seconds.
Also, we can see that the running times of bubble sort for the same input size are high by comparing this graph with all the previous graphs. Hence, if we need to use bubble sort then always implement the optimized algorithm.

The graph which contains the plot of all algorithms is as follows:



The graph which contains all plots has input of 9 random lists of size 100,200,300,400,500,600,700,800 and 900.
From the graph we can deduce that algorithms which have time complexity O(nlog(n)) are performing better as compared to the algorithms with time complexity $O(n^2)$.

- Merge sort is slightly faster than the heap sort for larger size data as we can see from the graph that when the input size of list is 700,800 and 900.
- Quicksort does not swap elements that are already in order, which is unnecessary and is done in heap sort. Hence, quick sort outperforms heap sort. From the graph we can see the running time is very less for any size of input for quick sort, just very slightly above 0.000 seconds.
- Also, the locality of reference in quick sort is better than merge sort. As a result, it outperforms merge sort as well. From the graph we can see the running time is very less for any size of input for quick sort, just very slightly above 0.000 seconds.
- Although insertion sort and bubble sort have same complexity $O(n^2)$. The insertion sort is little over twice as efficient as the bubble sort. As it can be seen from the graph.
- From the above graph we can observe bubble sort is the worst performing algorithm even though we have implemented its optimized version. Since, it does more element assignments than insertion sorting.

Inputs

As mentioned earlier there are two inputs in the algorithm implementation code.
1st input is user choice array of numbers. (Please note that it is not hardcoded)
For my testing purpose the input array used is as follows:
234,123,1,2,3123,4253,12,312,5243,456,52343,3,3242,546765,3,21321,2342354
The 2nd input is used for plotting the graphs and it is generated randomly.

Outputs

I have used Pycharm CE for executing my program and the output seen in the Run prompt is as follows:

/Users/pb/PycharmProjects/DAA/venv/bin/python /Users/pb/PycharmProjects/DAA/Algorithms.py
The sorting Algorithms are as follows:

1. Mergesort
2. Heapsort
3. Quicksort (Using median of 3)
4. Insertion sort
5. Bubble sort

Enter numbers separated by a comma:
234,123,1,2,3123,4253,12,312,5243,456,52343,3,3242,546765,3,21321,2342354

The sorted numbers by Mergesort are as follows:
[1, 2, 3, 3, 12, 123, 234, 312, 456, 3123, 3242, 4253, 5243, 21321, 52343, 546765, 2342354]

The sorted numbers by Heapsort are as follows:
[1, 2, 3, 3, 12, 123, 234, 312, 456, 3123, 3242, 4253, 5243, 21321, 52343, 546765, 2342354]

The sorted numbers by Quicksort (Using median of 3) are as follows:
[1, 2, 3, 3, 12, 123, 234, 312, 456, 3123, 3242, 4253, 5243, 21321, 52343, 546765, 2342354]

The sorted numbers by Insertion sort are as follows:
[1, 2, 3, 3, 12, 123, 234, 312, 456, 3123, 3242, 4253, 5243, 21321, 52343, 546765, 2342354]

The sorted numbers by Bubble sort are as follows:
[1, 2, 3, 3, 12, 123, 234, 312, 456, 3123, 3242, 4253, 5243, 21321, 52343, 546765, 2342354]

The input size vs running time graph is plotted for all algorithms

100 input list numbers got sorted by merge sort in time 0.000926
200 input list numbers got sorted by merge sort in time 0.001832
300 input list numbers got sorted by merge sort in time 0.0068
400 input list numbers got sorted by merge sort in time 0.005912

500 input list numbers got sorted by merge sort in time 0.005675
600 input list numbers got sorted by merge sort in time 0.005058
700 input list numbers got sorted by merge sort in time 0.005644
800 input list numbers got sorted by merge sort in time 0.003786
900 input list numbers got sorted by merge sort in time 0.006427

100 input list numbers got sorted by heap sort in time 0.000707
200 input list numbers got sorted by heap sort in time 0.002048
300 input list numbers got sorted by heap sort in time 0.002827
400 input list numbers got sorted by heap sort in time 0.004305
500 input list numbers got sorted by heap sort in time 0.004287
600 input list numbers got sorted by heap sort in time 0.00521
700 input list numbers got sorted by heap sort in time 0.007585
800 input list numbers got sorted by heap sort in time 0.008652
900 input list numbers got sorted by heap sort in time 0.007723

100 input list numbers got sorted by quick sort (Median 3) in time 0.000163
200 input list numbers got sorted by quick sort (Median 3) in time 0.000326
300 input list numbers got sorted by quick sort (Median 3) in time 0.000523
400 input list numbers got sorted by quick sort (Median 3) in time 0.001324
500 input list numbers got sorted by quick sort (Median 3) in time 0.001113
600 input list numbers got sorted by quick sort (Median 3) in time 0.001159
700 input list numbers got sorted by quick sort (Median 3) in time 0.001496
800 input list numbers got sorted by quick sort (Median 3) in time 0.002159
900 input list numbers got sorted by quick sort (Median 3) in time 0.001759

100 input list numbers got sorted by insertion sort in time 0.000719
200 input list numbers got sorted by insertion sort in time 0.003012
300 input list numbers got sorted by insertion sort in time 0.007003
400 input list numbers got sorted by insertion sort in time 0.011514
500 input list numbers got sorted by insertion sort in time 0.018923
600 input list numbers got sorted by insertion sort in time 0.028158
700 input list numbers got sorted by insertion sort in time 0.04045
800 input list numbers got sorted by insertion sort in time 0.048473
900 input list numbers got sorted by insertion sort in time 0.059337

100 input list numbers got sorted by bubble sort in time 0.002768
200 input list numbers got sorted by bubble sort in time 0.010228
300 input list numbers got sorted by bubble sort in time 0.022278
400 input list numbers got sorted by bubble sort in time 0.041867
500 input list numbers got sorted by bubble sort in time 0.070933
600 input list numbers got sorted by bubble sort in time 0.090815
700 input list numbers got sorted by bubble sort in time 0.133267
800 input list numbers got sorted by bubble sort in time 0.16043

900 input list numbers got sorted by bubble sort in time 0.208393

The graph plot of a specific sorting algorithm is as follows:
The input size vs running time graph for Merge sort.

100 input list numbers got sorted by Merge sort in time 0.000648
200 input list numbers got sorted by Merge sort in time 0.001271
300 input list numbers got sorted by Merge sort in time 0.001492
400 input list numbers got sorted by Merge sort in time 0.001921
500 input list numbers got sorted by Merge sort in time 0.002517
600 input list numbers got sorted by Merge sort in time 0.003064
700 input list numbers got sorted by Merge sort in time 0.004166
800 input list numbers got sorted by Merge sort in time 0.003984
900 input list numbers got sorted by Merge sort in time 0.004329

The graph plot of a specific sorting algorithm is as follows:
The input size vs running time graph for Heap sort.

100 input list numbers got sorted by Heap sort in time 0.000466
200 input list numbers got sorted by Heap sort in time 0.001136
300 input list numbers got sorted by Heap sort in time 0.001917
400 input list numbers got sorted by Heap sort in time 0.002582
500 input list numbers got sorted by Heap sort in time 0.003223
600 input list numbers got sorted by Heap sort in time 0.004278
700 input list numbers got sorted by Heap sort in time 0.005173
800 input list numbers got sorted by Heap sort in time 0.00603
900 input list numbers got sorted by Heap sort in time 0.006552

The graph plot of a specific sorting algorithm is as follows:
The input size vs running time graph for Quick sort(Median 3).

100 input list numbers got sorted by quick sort (Median 3) in time 0.00016
200 input list numbers got sorted by quick sort (Median 3) in time 0.00031
300 input list numbers got sorted by quick sort (Median 3) in time 0.000495
400 input list numbers got sorted by quick sort (Median 3) in time 0.000665
500 input list numbers got sorted by quick sort (Median 3) in time 0.000974
600 input list numbers got sorted by quick sort (Median 3) in time 0.001097
700 input list numbers got sorted by quick sort (Median 3) in time 0.001421
800 input list numbers got sorted by quick sort (Median 3) in time 0.001541
900 input list numbers got sorted by quick sort (Median 3) in time 0.00212

The graph plot of a specific sorting algorithm is as follows:
The input size vs running time graph for Insertion sort.

100 input list numbers got sorted by insertion sort in time 0.00063
200 input list numbers got sorted by insertion sort in time 0.002881
300 input list numbers got sorted by insertion sort in time 0.006177

400 input list numbers got sorted by insertion sort in time 0.011306
500 input list numbers got sorted by insertion sort in time 0.016372
600 input list numbers got sorted by insertion sort in time 0.023966
700 input list numbers got sorted by insertion sort in time 0.03225
800 input list numbers got sorted by insertion sort in time 0.045683
900 input list numbers got sorted by insertion sort in time 0.055968

The graph plot of a specific sorting algorithm is as follows:
The input size vs running time graph for Bubble sort.

100 input list numbers got sorted by bubble sort in time 0.00248
200 input list numbers got sorted by bubble sort in time 0.010596
300 input list numbers got sorted by bubble sort in time 0.022622
400 input list numbers got sorted by bubble sort in time 0.042536
500 input list numbers got sorted by bubble sort in time 0.063147
600 input list numbers got sorted by bubble sort in time 0.092647
700 input list numbers got sorted by bubble sort in time 0.122573
800 input list numbers got sorted by bubble sort in time 0.165491
900 input list numbers got sorted by bubble sort in time 0.214228

Process finished with exit code 0

References:

1. https://stackabuse.com/sorting-algorithms-in-python/
2. https://www.geeksforgeeks.org/introsort-or-introspective-sort/
3. https://www.cprogramming.com/tutorial/computersciencetheory/sortcomp.html
4. https://www.java-tips.org/java-se-tips-100019/24-java-lang/1896-quick-sort-implementation-with-median-of-three-partitioning-and-cutoff-for-small-arrays.html
5. https://www.quora.com/What-are-the-advantages-and-dis-advantages-of-bubble-sorting
6. http://www-cs-students.stanford.edu/~rashmi/projects/Sorting.pdf
7. https://rosettacode.org/wiki/Sorting_algorithms/Quicksort
8. Review notes.pdf provided by Ma'am for 1st exam review.